Naval Research Laboratory
Washington, DC 20375-5320

NRL/MR/5540--02-8629

Towards a Methodology and Tool for the Analysis of Security-Enhanced Linux Security Policies

MYLA ARCHER
ELIZABETH LEONARD

Center for High Assurance Computer Systems
Information Technology Division

MATTEO PRADELLA

CNR CESTIA
Politecnico di Milano
Milano, Italy

August 16, 2002

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave Blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	August 16, 2002	

4. TITLE AND SUBTITLE

Towards a Methodology and Tool for the Analysis of Security-Enhanced Linux Security Policies

5. FUNDING NUMBERS

WU - 55-6601-B2

6. AUTHOR(S)

Myla Archer, Elizabeth Leonard, and Matteo Pradella*

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
4555 Overlook Avenue, SW
Washington, DC 20375-5320

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL/MR/5540--02-8629

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

DARPA (ATO)
3701 North Fairfax Drive
Arlington, VA 22203

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

*CNR CESTIA, Politecnico di Milano, Milano, Italy

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

Security-Enhanced (SE) Linux, released by NSA in January 2001, is a version of Linux that adds security features by superimposing the Flask architecture on its kernel. In this architecture, a security server decides whether to grant particular subjects (i.e., processes) permissions to particular objects. The decisions are based on a combination of type enforcement (TE), role-based access control (RBAC), and multilevel security (MLS) rules. SE Linux includes a policy language for defining these parts of the security policy. Because the policy language forces detailed specification of which access permissions may be granted, the relation of a policy definition to high level security goals is not obvious. The length and complexity of policy definitions (thousands of rules is typical) precludes proving a high level security property by inspection. For policy analysis, tool support is clearly needed. To be effective in practice, this tool support needs to be usable by members of the open source software community. This paper reports progress made towards adapting the theorem-proving tool TAME to the analysis of SE Linux security policies, and making it usable to open source developers.

14. SUBJECT TERMS

SE Linux
Open source software
Operating systems security
Security policy analysis
Verification
Theorem proving
Automated deduction

15. NUMBER OF PAGES

45

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std 239-18
298-102

CONTENTS

Towards a Methodology and Tool for the Analysis of Security-Enhanced Linux Security Policies *

Myla Archer Elizabeth Leonard

Code 5546, Naval Research Laboratory, Washington, DC 20375 USA
E-mail: {archer,leonard}@itd.nrl.navy.mil

Matteo Pradella

CNR CESTIA, Politecnico di Milano, Milano ITALY
E-mail: pradella@elet.polimi.it

Security-Enhanced (SE) Linux is a version of Linux with additional security features. The initial version of SE Linux was released by NSA in January, 2001. The additional security features are incorporated into Linux by superimposing the Flask architecture on its kernel. This architecture includes a security server that makes decisions as to whether particular subjects (i.e., processes) may be granted particular permissions to particular objects. The decisions are made in accordance with a security policy that is a combination of a type enforcement (TE) policy, a role-based access control (RBAC) policy, and, optionally, a multilevel security (MLS) policy. Associated with SE Linux is a policy language in which these various parts of the security policy can be defined. An important issue with respect to a given policy is whether it achieves the high level goals for which it is intended. Because the policy language is aimed at a detailed specification of which access permissions may be granted, the relation of a given policy to its high level goals is not obvious. Because policy specifications tend to be very detailed and complex (e.g., the specification of the example security policy accompanying the SE Linux release is over 80 pages long), establishing that the policy achieves any particular high level goal cannot simply be done by inspection. For the analysis of policies, tool support is clearly needed. To make it practical for members of the open source software community to analyze security policies that they define, this tool support needs to be usable by members of this community. This paper reports progress made towards adapting the tool TAME, a PVS interface designed to support specification and analysis of automata in a user-friendly manner, to the analysis of SE Linux security policies. It describes a general approach to modeling an SE Linux security policy as an automaton, expressing its security goals as automaton properties, and applying TAME. It also describes the progress made in applying this approach to a subset of the example security policy in the SE Linux release.

1. Introduction

In January 2001, the National Security Agency issued its initial release of Security-Enhanced (SE) Linux, a version of Linux with added security features. The purpose of the SE Linux release is to be an initial step towards providing the means for open source software developers to make their software more secure.

* This work is funded by DARPA.
Manuscript approved July 25, 2002.

Accompanying the SE Linux release is a policy language and an example security policy defined using this language. The policy language allows a developer to specify three aspects of the policy: a type enforcement (TE) component, a role-based access control (RBAC) component, and a multilevel security (MLS) component. It is hoped by NSA that open source software developers will accompany their software with policies that ensure that their applications meet appropriate security goals.

To provide for policy enforcement in SE Linux, the Flask architecture [10] is superimposed upon the Linux kernel. In this architecture, a *security server* determines from the definition of a policy whether any given request by a process for a particular permission to a particular object may be granted. A policy described in the SE Linux policy language is compiled by a program `checkpolicy` into an internal form understandable to the security server.

With respect to policy correctness, there are at least three important questions to ask about any individual policy:

- Is the policy compiled correctly? I.e., is its internal form equivalent to its definition?
- Is the policy correctly enforced by the security server?
- Does the policy, as deduced from the semantics of its definition in the policy language, meet a given set of security goals?

Answering the first two questions requires understanding the internal form of a policy and verification of the security server code. The project described in this paper addresses the third question, which one might *a priori* expect to be independent of details of the actual SE Linux code.

Any policy defined in the SE Linux policy language will be intended to accomplish a set of security goals. The documentation accompanying the SE Linux release describes a set of eight security goals for the example policy in the release [9]. However, it is not immediately obvious that a given policy achieves particular security goals. Because the policy language is designed to specify (mandatory) access control at a detailed level, the analysis required to determine whether the goals are achieved is too complex to do by inspection. To perform such an analysis, tool support is clearly required. To encourage open source developers to analyze the policies they define with respect to their security goals, this tool support needs to be usable by software developers.

The project described in this paper has two goals. Ultimately, the goal is to produce a methodology and associated tool support permitting developers to analyze their policies. As intermediate goal is to first demonstrate this methodology and tool support on the example security policy accompanying the SE Linux release. The tool support is to be based on TAME, a specialized interface to PVS that provides user-friendly support for specifying and proving properties of automata models.

One essential requirement that must be met before either of our above goals can be achieved is to have a precise semantics for the SE Linux policy language; without such a semantics, it is impossible to prove any properties of a policy defined in the language. Because the description provided in the SE Linux documentation is very informal, relying heavily on examples, the initial phase of the project has been to formalize this semantics.

The policy definition accompanying the SE Linux release is extremely complex, covering over 80 pages, even with macros to allow individual rules to cover many cases. Because our intermediate goal—demonstrating the methodology and tool on the example policy—is essentially a feasibility study, we have chosen a subset of the full example policy on which to perform the initial demonstration.

This paper is organized as follows. Section 2 describes our overall approach, which has six phases. The next six sections describe our progress in terms of these six phases towards developing a general methodology and applying it to an example. In particular: Section 3 describes our formalization of the policy language semantics; Section 4 describes our general approach to representing a security policy with a state machine; Section 5 discusses the representation of security goals as state machine properties in general, and describes how the security goals stated in [9] can be represented as state machine properties; Section 6 describes the policy subset we have chosen for our initial study; Section 7 describes how TAME can be used to check properties of a security policy; and Section 8 describes our plans for technology transfer. Finally, Section 9 discusses some of the difficulties and questions that have arisen in developing our state machine model, and Section 10 presents our conclusions.

2. Approach

Our approach to developing a methodology with tool support that can be used by developers in the open source software community has six stages:

1. Develop a precise formal description of the policy language and its semantics.
2. Develop a general state machine model for the system to which a security policy is to be applied.
3. Develop a method by which to express policy goals as properties of the state machine model.
4. Choose a small subset of the policy accompanying the SE Linux release that will serve as an initial example application for our methodology.
5. Use TAME to analyze the reduced model with respect to the policy goals for the SE Linux release.
6. Technology transfer: Document the methodology and adapt TAME appropriately to make use of the methodology and TAME feasible for open source developers without deep knowledge of mechanical theorem proving.

Our progress in following this approach is detailed below.

3. The Policy Language

The SE Linux security policy language is described in [8], part of the documentation accompanying the SE Linux release. The language description in [8] is somewhat informal, and is mostly given by example. Some of the language constructs are not fully defined in [8]; however, most of the constructs used in the example policy accompanying the release have reasonably complete descriptions.

This section summarizes the language constructs mentioned in [8] that relate to TE and RBAC policies, and gives our understanding of their purposes. Our attempt at a formal description of the syntax and semantics of the various constructs is given in Appendix A.

The SE Linux policy language has four kinds of statements: *declarations*, *rules*, *constraints*, and *assertions*. *Declarations* include *role declarations* and *type declarations*. *Rules* include *access vector rules*, which govern decisions made by the security server about access requests, and *transition rules*, which govern possible role changes of an object and type-enforcement type assignments to newly created objects. *Constraints* constrain the manner in which various access permissions can be applied to various objects. *Assertions* are statements about whether or not certain kinds of access permissions are ever allowed by the policy. While the declarations, rules, and constraints are enforced by the security server at run time, the assertions are checked by the policy compiler `checkpolicy` at policy compile time. Thus, assuming `checkpolicy` works correctly, the assertions can be used as simple properties of the security policy that are available as lemmas in the proof of more complex properties (such as invariants capturing high level security goals).

Each language statement consists of a keyword (e.g., "allow" for the most typical access vector rules) followed by arguments that are expressed by using other language elements such as *type names*, *role names*, *object classes*, *attributes*, and *permissions*. The particular sets of representatives of these elements can depend on the particular policy being defined (and the particular Linux configuration for which it is being defined—e.g., the particular kernel modules present). The sets tend to be quite large. In the example policy with the SE Linux release, there are 3 role names, 28 object classes, 22 attributes, 115 permissions, and 253 type names of which 21 are parameterized—meaning there is an unbounded number of type names. The size problem is handled by the definition of macros for sets of permissions, sets of attributes, sets of object classes, etc. These macros can be used in place of individual arguments in the syntax for the statements described in Appendix A.

4. A General State Machine Model

A state machine model can be specified by defining 1) a set of *states*, 2) an *initial state* (or set of initial states), and 3) a set of possible *state transitions*.

Modeling the workings of an SE Linux security policy requires an abstract state machine that captures the security-relevant features of the system subject to the policy.

Modeling states. One role of the security policy is to define the mandatory access control policy of the system by determining the kinds of access which the *subjects* (i.e., processes) of the system may have to the system *objects* (e.g., file objects, process objects, etc.). In SE Linux, each object has an associated *security context* consisting of a set of features including a *(TE) type*, a *user*, a *role*, and if relevant, a *multi-level security level.* Since access control decisions are based on security contexts, a state in the state machine model must include representations of the objects of the system, and include a security context in the representation of each object. This information can be kept in a state variable `objects`.

Typical security goals involving changes to an object only consider whether or not the object could have been changed. To model changes to file objects, we associate an abstract "content" in the form of a natural number with each object, which starts at 0 and is incremented each time a change could have happened.

The security policy may contain information on when to audit certain requests for permissions. In this case, the state must keep track of audit information. Conceptually, this information is kept in a special (file) object. However, we expect to model audit information as a separate state variable `audit` in order to simplify both the notion of "content" of a file object and the retrieval of audit information from the state. The exact representation of this information may change if a description is provided of how auditing in SE Linux is expected to be used; there currently seems to be no such description.

Modeling the initial state. The initial state chosen for the state machine model will vary with the subset of the SE Linux security policy that is being modeled. In our initial subset (see Section 6), we expect to model the system after it has been initialized and ready for a user to log in. For this subset, the initial state will include among its objects the login daemon, standard system software executables, etc. To model how the security policy controls system initialization, a different, more primitive initial state would be needed.

Modeling state transitions. Security contexts can be affected by *transitions* such as *type transitions* and *role transitions.* A further role of the security policy beyond determining access control decisions is to define the permissible transitions. Therefore, the state transitions in the state machine model must reflect both access control decisions and possible modifications to security contexts.

In the Flask architecture (see [10]), transitions of the system are triggered by actions called "user requests". These requests are made to the "object manager", which consults the security server before handling them. It seems clear that the transitions in the state machine model must correspond to the user requests. However, the term "user request" can be used to describe requests at

various levels. For example, a shell level command might be a user request; at a lower level, so might a system call; at an even lower level, so might a request to be granted a permission. A combination of criteria led us to decide that the system call is the best choice of user request to associated with an individual state transition in our model. First, transitions should have an easily described effect on the state of our model. Second, there should be a relatively small and well-defined set of distinct requests. Individual permission checks do not meet the first criterion, and neither permission checks nor shell level commands meet the second.

The documentation of the security policy in the SE Linux release provides details as to the permission checks needed for individual system calls, providing a natural way to tie the access decisions defined by the security policy to the system calls. One complication, however, is that when SE Linux processes a system call and some necessary permission is denied, any remaining permissions checks are skipped. Exactly which ones are skipped can only be determined from looking at the kernel code [7]. To keep the abstract model of the system as simple as possible, we instead perform the full set of permission checks. This can make the model diverge from the actual system if any permission check with a side effect is done in the model when it would not be done in the system. However, since the only side effects of permission checks appear to be accumulation of audit information, and since this audit information does not play any obvious role in achieving the security goals in [9], we consider this divergence to be normally harmless.

We have written pseudo-code to describe the arguments to and effects of those system calls that are relevant to the policy and system subset (described below in Section 6) that we have chosen for our initial example. Details of our method for selecting the subset of system calls and for arriving at our pseudo-code representations for these system calls are provided in Appendix C.

Determining the effects of the various system calls is straightforward from the Linux documentation. However, the effect of an automaton action based on a system call depends on whether the necessary permissions are granted. Requiring that all the needed permissions be granted places a precondition on the action. The complete transition function of the automaton requires computation of these preconditions based on the security policy specification. Because of the complexity of this specification, great care must be taken in organizing this computation for efficiency.

5. Expressing Security Goals as State Machine Properties

It is preferable, when possible, to capture policy goals as invariant properties of the state machine. These invariant properties have two flavors: *state invariants* (properties of all reachable states) and *transition invariants* (properties of the pairs of states in all reachable transitions). The reason for preferring

invariant properties is that they are simpler to establish with a theorem prover than other types of properties. As will be discussed below, many of the security goals in [9] for the policy in the SE Linux release can be expressed in terms of state or transition invariants. However, no cut-and-dried general methodology for expressing security goals as state machine properties can be given. The best one can hope for is some general guidelines and a set of examples to use as models.

Policy goals formulated as state machine properties must be expressed in terms of the state variables of the state machine used to model the workings of the policy. Thus, it is necessary to develop the state machine model for the policy before formalizing its security goals. The advantage to formalizing the goals is that it makes the intent of the goals unambiguous. For example, one of the goals for the SE Linux release is to "protect the integrity of the kernel". A precise formal version of this goal would make clear what is meant by integrity, and would clarify such issues as whether anything may change the kernel (e.g., whether a new module can be added).

The eight high level security goals described in [9] for the policy in the SE Linus release are as follows:

1. Control various forms of raw access to data.
2. Protect the integrity of the kernel.
3. Protect the integrity of the system software, system configuration information, and system logs (i.e., only the system administrator can modify the system software).
4. Confine potential damage from exploitation of a flaw in a process that requires privileges.
5. Protect privileged processes from executing malicious code.
6. Protect the administrator role and domain from being entered without user authentication.
7. Prevent ordinary user processes from interfering with system processes or administrator processes.
8. Protect users and administrators from the exploitation of flaws in the `netscape` browser by malicious mobile code.

(Except for some paraphrasing in items 3 and 4 above, this list is quoted verbatim from [9].) Below, we give some examples of how many of these goals can be expressed in terms of state invariants or transition invariants:

- Goal 1 can be expressed in terms of state invariants of the form "In any state, only subjects having security context i can have permission j to objects having security context k", where i, j, and k need to be specified by the policy designer. These invariants may need to be supplemented by invariants of the

form "Subject <name> can never have security context i" or "A subject with initial security context c can never acquire security context i". The latter can be expressed as a state invariant if object histories are maintained as part of the current value of the object in the state machine model.

- Goals 2 and 3 can both be expressed in terms of transition invariants of the form "If an action changes the content of an object with security context i, then the action must have resulted from a successful request of a subject with security context j". Goal 4 can be expressed in terms of similar transition invariants, of the form "If an action changes the content of an object with security context i, then the action must have resulted from a successful request of a subject with security context j_1, ... , or j_n", where the security contexts i covers the objects which one does not want a subject with security context k $\notin \{j_1, \ldots, j_n\}$ to be able to affect.
- Goal 6 can be expressed as the transition invariant "If an action results in the subject entering a security context corresponding to an administrator role or domain, then the subject must have had its authentication value set in the prestate of the transition". This requires that process objects (potential subjects) have as part of their representation in the model an associated value indicating their authentication status.
- Goal 7 can be expressed in terms of transition invariants of the form "If an action changes the state of a system or administrator process object, then it is not the result of a request by an ordinary user process object".

Note that it generally takes several invariants to express a security goal. The invariants sketched above clearly cannot be stated more precisely until the variables and actions in the state machine model have been specified. Moreover, the ability to capture some of the goals, e.g., Goals 6 and 7, in terms of the abstract state machine model relies on the model being sufficiently detailed. In particular, the invariants for Goal 7 require a reasonable abstract representation of the state of a process object to be included in the model, beyond the integer to represent content that is sufficient to capture changes in the state of a file object.

Because the goals are stated in such vague terms, precise details such as the values to assign to the security context parameters cannot really be determined. Such details would have to be provided by the designers of the security policy, who no doubt have a more precise notion of the intentions of the various goals.

6. Choosing an Initial Policy Subset

To demonstrate the feasibility of our approach, we model a small but useful subset of the example policy which accompanies the SE Linux release. Our chosen subset is the portion of the original system necessary for a single user to operate on a minimal level. We assume that the system has already been booted and properly

initialized. The subset includes the procedures for login and password changing as well as the usual operations on files and directories. Programs not necessary to the basic functioning of the operating system, such as the X server, the gnome pty helper, mail, printing, mount, logrotate, at, and cron, are eliminated. Kernel module utilities and System V inter-process communication are also excluded from the subset. The SE Linux classes associated with Linux capabilities and security aware applications have also been eliminated. Elimination of a particular program was done by removing the type declarations associated with the program and all references to those types in rules. Particular classes were removed by eliminating all rules that refer to those classes.

The subset also abstracts away some of the detail related to hardware interaction. Programs such as the card manager, the advanced power management system, and the console mouse server are exluded from the subset. The hardware devices, which are files in the system, are retained in the subset because some of them are needed for the ioctl system calls.

All networking capabilities are excluded from the subset, as are the network communications programs rlogin, rpc, rsh, ssh, and tcp. All sockets are excluded, except for Unix datagram sockets, which are used for inter-process communication. Also excluded are Netscape, NFS, and programs which help facilitate networking such as ypbind, ifconfig, inetd, and NIS.

The set of system calls used to form the set of transitions in the model (see Appendix C) is used to further reduce the example policy. These system calls do not require all of the possible permissions for the classes of objects remaining in the system. Thus, all rules relating to permissions unused by the system calls are eliminated.

The reduced system is significantly smaller than the original example policy, as shown by the statistics below.

	SE Linux Example Policy	Reduced System Policy
Number of TE files	53	23
Number of allow rules	708	234
Number of types	253	96
Number of object classes	28	12
Number of permissions	115	34

For a complete description of what was retained in the reduced policy see Appendix B.

The remaining subset is still large enough to allow the formulation of properties related to most of the security goals from [9] in Section 5. The reduced system still retains the various hardware devices and processes that need to access them, allowing properties related to Goal 1 to be analyzed. Though we assume

the system is booted, we retain the files containing the code for booting and can thus check for access to these files. We also retain rules involving transitions to the type of the initialization process, allowing us to analyze whether there are any processes that can improperly transition to the initialization process. Both of these inclusions make it possible to formulate properties related to Goal 2. System software, configuration information, and log files remain in the subsystem, as do some programs that need access to those various types of files, allowing Goal 3 to be studied. Properties related to Goals 4 and 5 can be formulated because the reduced system retains the login process, which is privileged. The reduced system contains the administrator role and domain, as well as the login and newrole programs, which are necessary for Goal 6 to be analyzed. Properties related to Goal 7 can be formulated because there are user, system, and administrator processes remaining in the system.

After the reduced policy has been analyzed to see if it satisfies the security goals, eliminated programs could be reintroduced and the resulting policy analyzed. Some portions of the example policy, such as that associated with booting and initializing the system, could be independently analyzed.

7. Analyzing the State Machine Model of a Policy with TAME

TAME [4,1] is an interface to PVS designed to simplify the specification of automata models and proofs of automata properties, especially invariant properties. To specify an automaton, the user fills in the TAME template, providing the information shown in Figure 1. Auxiliary definitions are also usually needed to support the information the user provides in the template. For example, if there are state variables whose types are more complex than simply boolean or

Template Part	User Fills In	Remarks
`actions`	Declarations of non-time-passage actions	`actions` is a PVS datatype
`MMTstates`	Type of the "basic state" representing the state variables	Usually a record type
`OKstate?`	An arbitrary state predicate restricting the set of states	Default is `true`
`enabled_specific`	Preconditions for all the non-time-passage actions	`enabled_specific(a,s)` = specific precondition of action `a` in state `s`
`trans`	Effects of all the actions	`trans(a,s)` = state reached from state `s` by action `a`
`start`	State predicate defining the initial states	Preferred forms: `s = ...` or `s = (# basic := basic(s) WITH ... ... #)`
`const_facts`	Predicate describing relations assumed among the constants	Default is `true`

Figure 1. Information needed to fill in the TAME template.

numeric, these types must be defined in type declarations. The major state variable `objects` in our general model for SE Linux describes a set of objects with associated characteristics. The most convenient way to represent `objects` in PVS is as a function that takes a unique object identifier (e.g., a unique path name) and returns a record of its characteristics (e.g., user, TE type, role, MLS level, abstract content, etc.). This very complex type for `objects` requires several supporting type declarations. Other auxiliary definitions that may be needed in a given specification are function definitions and axioms. Auxiliary function definitions are especially useful in "layering" the definition of the transition function `trans` and the precondition function `enabled_specific` so that these definitions can be expanded only to the extent needed in reasoning about individual state transitions.

TAME also provides a standard set of steps for reasoning about invariant properties of automata. The most commonly used steps are shown in Figure 2.

TAME Strategy	Purpose
AUTO_INDUCT	Set up a structural induction proof
DIRECT_PROOF	Set up a non-induction proof
APPLY_SPECIFIC_PRECOND	Introduce the specified precondition
APPLY_IND_HYP	Apply the inductive hypothesis
APPLY_INV_LEMMA	Apply an invariant lemma
APPLY_LEMMA	Apply any general lemma
SUPPOSE	Do a case split and label the cases
TRY_SIMP	Try to complete the proof automatically

Figure 2. Most commonly needed TAME proof steps in invariant proofs.

The TAME proof steps have proved sufficient to prove invariant properties in a wide set of applications; see, for example, [3,4,2]. In addition to the proof steps shown in Figure 2, an automatic proof strategy based on these steps has been developed for proving invariant properties of automata specified in the SCR toolset [5]. This automatic strategy has been used to prove properties of a moderate sized SCR example [6]. An associated analysis strategy helped in finding a counterexample to a property that was not an invariant. The layered nature of the representation of the transition function of the TAME representation of an SCR specification was used to advantage in designing the automatic proof strategy for SCR specificiations for efficiency. There is hope that analogous layering in the TAME representation of SE Linux models can be used to similar advantage.

The TAME step **APPLY_INV_LEMMA** in Figure 2 is used for applying other invariant lemmas in the course of the proof of the current invariant lemma. It is typical for desired system invariants to require auxiliary invariants in their

proofs. Experience has shown that for SCR automata, many of the needed auxiliary invariants are among the invariants automatically generated by the invariant generation algorithm in the SCR toolset. The automatic SCR proof strategy uses these automatically generated invariants when they are needed. In principle, it is not necessary to establish the automatically generated invariants using TAME; they can instead be included as axioms. As noted in Section 3, we hypothesize that the "assertions" in an SE Linux policy description can be used to support proofs of SE Linux policy invariants in analogy to the way the automatically generated invariants support proofs of more complex invariants in SCR.

8. Plans for Technology Transfer

Specifying the system model in TAME for a security policy defined in the SE Linux policy language requires extracting certain information from the policy definition and formulating this information appropriately for use in filling out the TAME template. The example policy subset that we are using as a feasibility study provides the basis for determining the information that needs to be extracted and how it will be used in TAME. Enabling practitioners such as open source software developers to create TAME models for their security policies will initially be done by means of documentation illustrated by our example. In the farther future it would be desirable to create a policy-to-TAME compiler.

To make TAME itself more accessible for developers, we expect to adapt both the specification and the proof support of TAME for SE Linux security policies. Because many aspects of TAME models of policies—such as the state variable `objects` and much of its type structure—will be common to all models, the TAME template can be refined into a specialized version for policy models that includes template features for these common parts. One common feature, the overall structure of the representation of the access control decisions of the security policy, can be used to advantage in designing efficient specialized proof steps for use in proving security policy properties. As discussed in Section 7, there is hope that a strategy that can prove many invariant properties automatically can be designed for SE Linux security policies in analogy to the automatic strategy for SCR specifications. If so, this would greatly increase the accessibility of TAME to software developers with no experience of theorem proving.

9. Discussion: Problems to Be Solved

To develop a model for the example SE Linux security policy (or a subset), several problems resulting from the size and complexity of the example policy must be solved. The first problem is to decide just what state information needs to be represented in the model. As noted in Section 5, the answer to this depends on the properties one wishes to prove of the policy.

The next problem is to find an appropriate representation of the mechanism used by the the security server to make access control decisions. Ideally, a layered representation could be found that would allow decisions to be made efficiently based on least information. The layering could then be used in organizing the data structure used by the transition function in the state machine model.

The example policy is expressed relatively compactly by means of macros. When these macros are expanded, the full example policy expands to several tens of thousands of allow rules. It is clearly necessary to imitate the macros used in the policy definition when creating the state machine model of the policy. Doing so may provide part of the answer to finding an appropriate layered representation for the access control decisions.

Already mentioned in Section 4 is the question of how to represent the effect on the state of user requests (system calls) whose associated permissions are only partly granted, when asking for a permission may lead to audit information being kept. Modeling this situation precisely can only be done by referring to the SE Linux source code, a thing to be avoided. Our proposed solution is to settle for a model that is not guaranteed to be precise about audit information.

10. Conclusions

The major difficulties associated with undertaking to prove properties of an SE Linux security policy as invariants of an appropriate state machine model arise from the sheer size of the policy description, which results from its low-level nature. Experience with applying mechanized support to the analysis of even relatively small systems has shown that specification errors are almost certain to be found even in system specifications that have been carefully scrutinized by many people prior to the mechanized analysis. Thus, one can expect that such an intricate policy as the one in the SE Linux release will leave some unexpected loopholes.

A more feasible approach to producing a policy known to have a desired set of high-level properties is to obtain it by compiling a high-level policy specification that is more amenable to analysis. A high-level policy definition language is definitely desirable [7].

In the absence of such a high-level language, it can be useful to model subsets of the full policy to search for loopholes. Provided these subsets include all of the policy relevant to some subset of Linux, any loophole found by this means will be a loophole in the full policy.

Acknowledgements

We thank James Kirby and Garth Longdon for their contributions to this project. We also gratefully thank Pete Loscocco for helpful discussions.

References

[1] M. Archer. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):139–181, 2000. Published Feb., 2001.

[2] M. Archer. Proving correctness of the basic TESLA multicast stream authentication protocol with TAME. In *Informal Proceedings of the Workshop on Issues in the Theory of Security (WITS'02)*, Portland, OR, Jan. 14–15 2002.

[3] M. Archer and C. Heitmeyer. Verifying hybrid systems modeled as timed automata: A case study. In *Hybrid and Real-Time Systems (HART'97)*, volume 1201 of *Lect. Notes in Comp. Sci.*, pages 171–185. Springer-Verlag, 1997.

[4] M. Archer, C. Heitmeyer, and E. Riccobene. Using TAME to prove invariants of automata models: Case studies. In *Proc. 2000 ACM SIGSOFT Workshop on Formal Methods in Software Practice (FMSP'00)*, Aug. 2000.

[5] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. on Softw. Eng.*, 24(11):927–948, Nov. 1998.

[6] James Kirby, Jr., M. Archer, and C. Heitmeyer. SCR: A practical approach to building a high assurance COMSEC system. In *Proc. 15th Annual Computer Security Applications Conference (ACSAC '99)*. IEEE Comp. Soc. Press, Dec. 1999.

[7] P. Loscocco. Private communication. NRL, Nov. 2001.

[8] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. Technical report, National Security Agency, Jan. 2, 2001.

[9] S. Smalley and T. Fraser. A security policy configuration for Security-Enhanced Linux. Technical report, National Security Agency, Jan. 2, 2001.

[10] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *Proc. of the Eighth USENIX Security Symposium*, pages 123–139, Aug. 1999.

Appendix

A. Semantics of the SE Linux Policy Language

Language statements in the SE Linux policy language come in four flavors: declarations, rules, constraints, and assertions. Assertions are checked against the others at policy compile time. The others are enforced by the SE Linux kernel. The statements described below are not the full set mentioned in the SE Linux domumentation, but do at least cover those actually found in the TE and RBAC policy descriptions for the example policy in the SE Linux release. The "Remarks" section in the description for a particular statement refer either to the uses of that statement in the example policy or to certain files of the policy descriptions. Caveat: the exact files and file contents in the policy description are likely to change over time. Our references are to the versions we captured on April 12, 2001.

A.1. Declarations

role declarations

```
Syntax:  role <role-name> types <types>

Examples:

 role system_r types initial_boot_t;

Remarks: There is only one example of role in the TE policy, in policy/init.te.
         Most role declarations are in the RBAC policy in policy/rbac.

Semantics:  Declares a new role <role-name> and specifies the set of types
            <types> that may be "entered" by a process in role <role-name>.

            Question: May a process "enter" a type by type_change?  By
            type_transition?  Both?  (I.e., just what "enter" means is
            not clear.)
```

type declarations

```
Syntax:  type <typename> <attributes>

Examples:

 type initial_boot_t, domain, privuser, privrole, privowner;

Remarks: There is supposedly an optional aliases argument, but we have
         found no uses of it.

         There are 253 type declarations in the TE policy, several of
         them parameterized.

Semantics:  Declares a new type <typename> which has the attributes
            listed in <attributes>.
```

A.2. Rules

According to the language description provided in [8], there are four kinds of access vector rules, namely: allow, auditallow, auditdeny, and notify. Only two of these, allow and auditdeny, are used in the TE policy. Quoting from [8]:

> These [four kinds of] rules define the corresponding access vectors returned by security_compute_av. If no rule is specified, then no permissions are returned in allowed, auditallow, or notify, and all permissions are returned in auditdeny. All permissions are always returned in the decided access vector, since the TE policy does not defer the computation of any permissions. ... Each access vector rule has a source type field, a target type field, a class field, and a permissions field.

Thus, in particular, any permission not explicitly allowed is denied (and audited).

Most of the rules we describe below can be used in more compact form by using sets of types, objects, and permissions as arguments where a single type, object, or permission is specified below. The interpretation of rules stated with such "plurals" is the same as the set of rules obtained by choosing each possible combination of "singular" arguments. An exception is the last <type> argument in a type_transition or type_change or type_member rule, which must name a single type (because it represents the necessarily unique default type to be assigned to a new or transformed object. There are several ways to represent sets: e.g., one can use 1) a comma-separated list in curly braces, 2) an attribute name to represent the set of types with that attribute, or 3) $\sim$ in front of an individual or a set to represent its complement.

allow rules

```
Syntax:  allow <type1> <type2>:<object class> <permission>

Examples:
 allow sysadm_t file_t:dir_file_class_set *;
 allow $1 netmsg_type:tcp_socket { connectto acceptfrom };

Remarks: There are 708 allow rules, most of them parameterized.  The
         parameter $1 in the example rule above can be any of the many
         types to which the can_network macro is applied.

Semantics:  Grant the requested permission.  Specifically, grant the
            permission <permission> to any subject of "source type" <type1>
            with respect to an object of "target type" <type2> and "class"
            <object class>.

            Example 1: allow domain init_t:process sigchld;

            Explanation given: This rule grants every domain the ability to
            send a SIGCHLD signal to init, so that init can reap every process.
```

```
Example 2: allow syslogd_t device_t:dir {read getattr access
                                         search add_name
                                         remove_name}

Explanation given: This rule grants syslogd the ability to access
/dev to replace /dev/log.


Example 3: allow syslogd_t devlog_t:sock_file create;

Explanation given: This rule grants syslogd the ability to create
the /dev/log socket file.
```

role allow rules

```
Syntax:  allow <role1> <role2>

Examples:
 allow system_r user_r;
 allow system_r sysadm_r;
 allow user_r sysadm_r;
 allow sysadm_r user_r;

Remarks: The above are the only (role) allow rules in the RBAC policy.

Semantics:  Allow an object with role <role1> to transition to role <role2>.
```

auditdeny rules

```
Syntax:  auditdeny <type1> <type2>:<object class> <permission>

Examples:
 auditdeny passwd_t initrc_var_run_t:file ~write;
 auditdeny system_crond_t initrc_var_run_t:file ~write;
 auditdeny local_login_t fixed_disk_device_t:blk_file ~{ getattr setattr };
 auditdeny local_login_t removable_device_t:blk_file ~{ getattr setattr };
 auditdeny local_login_t device_t:file_class_set ~{ getattr setattr };
 auditdeny local_login_t misc_device_t:file_class_set ~{ getattr setattr };
 auditdeny rshd_t etc_auth_t:dir ~search;
 auditdeny rshd_t etc_auth_t:file ~{read getattr};
 auditdeny sendmail_t initrc_var_run_t:file ~write;
 auditdeny $1_t domain:dir ~r_dir_perms;
 auditdeny $1_t domain:notdevfile_class_set ~r_file_perms;
 auditdeny $1_t initrc_var_run_t:file ~write;

Remarks: The above are the only uses of auditdeny in the TE policy.  The only
         values used for $1 above are "user", "sysadm", and "polyadm".

Semantics:  If the requested permission is denied, keep audit information
            about it.  Specifically, record any denial of the permission
            <permission> to any subject of "source type" <type1> with with
            respect to any object of "target type" <type2> and "class"
            <object class>.
```

```
Also, *do not* keep audit information for <type1>, <type2>, and
<object class> in regard to any other permission, unless there is
an explicit auditdeny rule for that permission.
```

type_change rules

```
Syntax:  type_change <type1> <type2>:<object class> <type3>

Examples:
 type_change sysadm_t user_tty_device_t:chr_file sysadm_tty_device_t;
 type_change user_t sysadm_tty_device_t:chr_file user_tty_device_t;
 type_change sysadm_t user_devpts_t:chr_file sysadm_devpts_t;
 type_change user_t sysadm_devpts_t:chr_file user_devpts_t;
 type_change user_t sshd_devpts_t:chr_file user_devpts_t;
 type_change $1_t tty_device_t:chr_file $1_tty_device_t;
 type_change $1_t rlogind_devpts_t:chr_file $1_devpts_t;

Remarks: The above are the only uses of type_change in the TE policy.
         $1 is the "domain_prefix" argument to the macro user_domain.
         The macro user_domain is applied only to the arguments "user",
         "sysadm", and "polyadm".

Semantics:  For any object of type <type2> and class <object class>, relabel
            the object with the type <type3> when a subject of type <type1>
            accesses the object.

            The "accesses" part is just a guess; to see what this
            really does, one presumably has to look at the code for
            the security server.
```

type_member rules

```
Syntax:  type_member <type1> <type2>:<object class> <type3>

Examples:
 type_member polyadm_t poly_t:dir poly_t;
 type_member $1_t poly_t:dir $1_home_t;

Remarks: The above are the only two uses of type_member in the TE policy.
         $1 is the "domain_prefix" argument to the macro user_domain.
         The macro user_domain is applied only to the arguments "user",
         "sysadm", and "polyadm".

Semantics:  ???  (Not described.)
```

type_transition rules

```
Syntax:  type_member <type1> <type2>:<object class> <type3>

Examples:
 type_transition $1 $2:process $3;
 type_transition $1 $2:dir $3;
 type_transition $1 $2:notdevfile_class_set $3;
 type_transition $1_t devpts_t:chr_file $1_devpts_t;
 type_transition $1_t devpts_t:chr_file $2_devpts_t;
 type_transition cardmgr_t tmp_t:chr_file cardmgr_dev_t;
 type_transition cardmgr_t device_t:lnk_file cardmgr_lnk_t;

Remarks: The above are the only uses of type_transition in the TE policy.

         $1, $2, and $3 in the first example are the arguments to the macro
         domain_auto_trans; in the second and third examples they are the
         arguments to the macro file_type_auto_trans.  Both of these macros
         are applied to many different sets of arguments in the policy.

         $1 in the fourth example is the domain_prefix argument to the
         can_create_pty macro, which is applied only to rlogind, sshd,
         and to the argument of the user_domain macro (which is only
         applied to "user", "sysadm", and "polyadm").

         $1 and $2 in the fifth example are the domain_prefix and
         other_domain arguments of the can_create_other_pty macro, which
         is used only within the gph_domain macro, applied to $1_gph and
         $1, where $1 is the argument to the gph_domain macro.  The
         gph_domain macro, in turn, is applied only to the argument of
         the user_domain macro, which is only applied to "user", "sysadm",
         and "polyadm".

Semantics:  If <object class> is "process", then the default type of any
            process object created from an executable of type <type2> by
            an object of <type1> will be <type3> rather than <type2>.

            If <object class> is a file object class, then the default type
            assigned to an object of class <object class> created in a
            directory of type <type2> by an object of type <type1> will be
            <type3> rather than <type2>.
```

role_transition rules

```
Syntax:  role_transition <role1> <type> <role2>

Examples:
 role_transition $1 $2 $3

Remarks: The above example, which occurs in the definition of the macro
         role_auto_trans, is the only use of role_transition in the RBAC
         policy; because role_auto_trans is not used at all, neither is
         role_transition.

Semantics:  The default role of a process created from an executable of
            type <type> by forking a process of role <role1> is <role2>
            rather than <role1>.
```

A.3. Constraints

constrain statements

```
Syntax:  constrain <object class> <permission> <expression>

Examples: constrain process transition ( u1 == u2 or t1 == privuser );
          constrain process transition ( r1 == r2 or t1 == privrole );
          constrain dir_file_class_set { create relabelto relabelfrom }
            ( u1 == u2 or t1 == privowner );

          constrain socket_class_set { create relabelto relabelfrom }
            ( u1 == u2 or t1 == privowner );

Remarks: The file "constraints" in the "policy" directory gives the
         following description of the syntax of constraints:

         # constrain class_set perm_set expression ;
         #
         # expression : ( expression )
         #       | not expression
         #       | expression and expression
         #       | expression or expression
         #       | u1 op u2
         #       | r1 role_op r2
         #       | t1 op t2
         #       | u1 op names
         #       | u2 op names
         #       | r1 op names
         #       | r2 op names
         #       | t1 op names
         #       | t2 op names
```

```
#
# op : == | !=
# role_op : == | != | eq | dom | domby | incomp
#
# names : name | { name_list }
# name_list : name | name_list name#
```

Semantics: The semantics from the file "constraints" for the first two examples is given as:

```
 # Restrict the ability to transition to other users
 # or roles to a few privileged types.
```

The semantics for the second two examples is given as:

```
 # Restrict the ability to label objects with other
 # user identities to a few privileged types.
```

The most probable interpretation is: A subject may apply the permission <permission> to an object of class <object class> only under the constraints denoted by the expression <expression>. In any expression, u1, r1, and t1 refer to the user, role, and type associated with the object before the permission is used, and u2, r2, and t2 refer to the user, role, and type that would be associated with the object after the permission were used. The symbol == can be used to denote membership in a set of types (e.g., possession of an attribute such as privuser).

A.4. Assertions

neverallow statements

Syntax: neverallow <types> <object> <permissions>

Examples:

```
neverallow ~{ kmod_t insmod_t rmmod_t ifconfig_t } self:capability sys_module;
neverallow ~klogd_t proc_kmsg_t:file ~stat_file_perms;
```

Remarks: Found only in policy/assert.te. The validity of these assertions is checked by the program "checkpolicy".

Semantics: Asserts that no permission in <permissions> is ever given to any entity of a type in <types> with respect to <object>.

B. The Reduced Policy

Roles. All three roles, **system_r**, **sysadm_r**, and **user_r**, are retained.

Types. The following types are retained in the reduced system. All other types are eliminated, along with any rules referring to those types.

$1_devpts_t	klogd_t	shell_exec_t
$1_home_t	klogd_tmp_t	shlib_t
$1_su_t	klogd_var_run_t	src_t
$1_tmp_t	ld_so_t	su_exec_t
$1_tty_device_t	lib_t	sulogin_exec_t
bin_t	local_login_t	sysadm_t
boot_runtime_t	local_login_tmp_t	sysctl_dev_t
boot_t	login_exec_t	sysctl_fs_t
chkpwd_exec_t	lost_found_t	sysctl_kernel_t
clock_device_t	ls_exec_t	sysctl_net_t
console_device_t	memory_device_t	sysctl_t
device_t	misc_device_t	sysctl_vm_t
devlog_t	newrole_exec_t	syslogd_exec_t
devpts_t	newrole_t	syslogd_t
devtty_t	no_access_t	syslogd_tmp_t
etc_aliases_t	null_device_t	syslogd_var_run_t
etc_auth_t	passwd_exec_t	tmp_t
etc_runtime_t	passwd_t	tty_device_t
etc_t	policy_config_t	unlabeled_t
file_labels_t	policy_src_t	user_t
file_t	poly_t	usr_t
fixed_disk_device_t	polyadm_t	utempter_exec_t
fs_t	proc_kcore_t	utempter_t
fsadm_exec_t	proc_kmsg_t	var_lib_t
fsadm_t	proc_t	var_lock_t
fsadm_tmp_t	psaux_t	var_log_t
getty_exec_t	ptmx_t	var_run_t
getty_t	random_device_t	var_spool_t
getty_tmp_t	removable_device_t	var_t
initrc_var_run_t	root_t	var_yp_t
kernel_t	sbin_t	wtmp_t
klogd_exec_t	security_t	zero_device_t

Classes. Classes retained in the subset are **dir**, **fd**, **filesystem**, **process**, **unix_dgram_socket**, **file**, **blk_file**, **chr_file**, **fifo_file**, **lnk_file**, **pipe**, and **sock_file**.

Permissions. The following are the permissions retained for each class. The permissions listed for **file** are also the permissions retained for classes **blk_file**, **chr_file**, **fifo_file**, **lnk_file**, **pipe**, and **sock_file**.

class	permissions
dir:	add_name read remove_name reparent rmdir search
fd:	create getattr inherit setattr
file:	access append create execute getattr ioctl link lock poll read rename setattr unlink write
filesystem:	associate
process:	entrypoint execute fork ptrace sigchld sigkill signal sigstop transition
unix_dgram_socket:	connect create read recvfrom recv_msg sendto send_msg write

Attributes. The attributes **netif_type**, **netmsg_type**, **node_type**, **port_type**, and **socket_type** are eliminated from the subset because all of the types having these attributes are not included in the subset.

C. System Calls

The initial TAME model of SE Linux is an abstract representation of a subset of SE Linux that omits certain system features such as those related to file system mounting, network operations, many I/O operations, and most kinds of sockets. The TAME model will represent this subset as a state machine with a set of actions representing user requests. The most logical choice for the level of these requests is the system call level, since the system calls are limited in number and can be used to represent the most security-relevant actions of system processes. Each system call results in a set of permissions being checked, directly or indirectly, by the security server, and will succeed only if all the permissions are granted.

This appendix contains pseudo-code descriptions of the system calls that will be modeled in the initial TAME model of SE Linux. The pseudo-code descriptions give details of the permissions asked for and of the effects of the system call if it succeeds. The system calls were selected by executing some simple user scenarios in Linux and determining which system calls resulted. The scenarios included:

1. log in; execute the passwd program; log out
2. log in; create a file; write something in the file; log out
3. log in; create & write a file; make the file executable; run the executable; log out

Certain system calls were eliminated from the results as either being probably irrelevant to our reduced model or to the TE policy:

1. Calls not covered or not clearly described by reference [8] (e.g. alarm, setuid, setrlimit).
2. File systems manipulation (e.g. mount).
3. System calls for security-aware applications.
4. Network system calls.
5. System V IPC calls (e.g. semaphores, shared memory).
6. Calls involving capabilities.

The system calls described in Section C.3 below that are marked with a bullet (•) are those that remained after the above selection process, and are intended to be definitely included in our initial example policy model. The remaining system calls described in Section C.3 were selected to include the most likely additional system calls from which we will choose any additions to our initial model.

Once the set of system calls to be modeled (or possibly modeled) was selected, a pseudo-code description of each was constructed using information from [8]. The information relevant to an individual system call was somewhat scattered in [8]; much of the information appears in tables, but the information in a

table is often incomplete. Sometimes, more than one table had to be consulted, and in most cases, text in the body of [8] provided additional detail. Other details of system calls were obtained directly from the Linux documentation.

In constructing the pseudo-code descriptions, some of the system call details were omitted—e.g., arguments that are irrelevant to the initial subset being modeled. In addition, the argument lists have sometimes been modified to break down a complex argument into its components, and to include a slot or slots for values being returned. In the individual system call descriptions below, we note these modifications.

Besides the high-level question of whether the pseudo-code representations of the system calls are correct, here are some questions that remain with respect to modeling the system calls in TAME:

1. What, exactly, happens when the set of permissions associated with a system call is checked? Are all permissions checked? This is relevant to modeling the collection of audit information; for example, if the permissions are checked in a certain order and if the system call fails as soon as one permission is denied, the fact that a later request that would be audited is also denied might be missed.
2. Should we model the "setuid" call in some way? It turns up a lot in our scenarios. But, the documentation implies that it is irrelevant to the security policy.
3. Can we eliminate any system calls (or additional details of their descriptions) from our initial model that we have not?

Several notational conventions are used in the individual system call descriptions. Some are from logic and set theory:

1. exists! for "there exists a unique"
2. $\{x|P(x)\}$ for "the set of all x such that P(x) holds"
3. "A U B" denotes union set of A and B
4. Sets are written uppercase, e.g. PID is the set of process pids.
5. x : TYPE is a type definition: it means that x has type TYPE.
6. "f(k) becomes t" denotes function value update to the function f, for argument k, to the value t.

C.1. Structure of the System

From a security enforcement point of view, the SE Linux kernel is ideally divided into three parts: a security server, object managers, and the remainder of the kernel. The object managers layer (from now on called OM) is in charge of the interaction between the security server and the other parts of the kernel (from now on called Ker). This appendix mainly describes this layer.

The OM subsystem is presented by means of its two main interfaces:

1. the Ker - OM interface, consisting of the function Ki.
2. the Security Server - OM interface, consisting of the three functions Ssav, Sst, and sid.

Caveat: audit mechanisms are not covered in this document.

Some Set Definitions

```
BOOL = {True, False}.

PID is the set of process pids.

FILENAME, FILEDESCR, FILESYSTEM, PATHNAME are the set of file names,
file descriptors, file systems, and pathnames, respectively.

SOCKET denotes the Unix datagram socket set and contains the standard
element any_socket.

KERNEL_CALL is the set of system calls.

SID is the set of security ids, i.e. references to security
contexts. SIDU is defined as SID added with the element "undef".

CLASS and PERMISSION are the sets of the security object classes and
permissions, respectively.

STRING is the set of generic strings.
```

C.2. Interface Function Signatures and Semantics

Kernel interface function Ki

```
In:  proc: PID,
     syscall: KERNEL_CALL
Out: res: BOOL with default False,
     act: set of KERNEL_CALL with default {}
```

Notes: The output "res" defines the policy check result: True means that the operation is permitted. The output "act" is a set of system calls that must be executed.

With a slight abuse of notation, sometimes "syscall" is written using a set of call names, e.g. {c1,c2}(x), with the obvious meaning that the pseudo-code appies to either c1(x) or c2(x). Square brackets are used for optional characters (in regular expression fashion) in the name of a "syscall" argument.

Security Server interface function Ssav

```
In:  source: SID,
     target: SID,
     obj: CLASS,
     perm: PERMISSION
Out: BOOL
```

Notes: This function is used to ask for permissions, and essentially is a kind of abstract version of security_compute_av ([8]).

Security Server interface function Sst

```
In:  source: SID,
     target: SID,
     obj: CLASS
Out: SIDU
```

Notes: This function is used for SID transitions or new SID requests. It is a kind of abstract version of security_transition_sid ([8]).

Role transitions occur with process transformation, i.e. after execve calls. The actual role is part of the security context, which is in a one-to-one correspondence with the SID. Therefore security context transitions are managed, in this abstract representation, by Sst.

Security Server interface function sid

```
In: obj: PID U FILENAME U FILEDESCR U FILESYSTEM U SOCKET U STRING
         U etc.
Out: SIDU
```

Notes: This function associates a SID to every Linux system item (Process, file, etc). More generally, it represents the "security state" of the whole system, and can be updated as a side effect of a Ki call.

C.3. The Kernel Interface Function Definition

C.3.1. Process Management

Close and Exit

Our signatures: exit(); •close(fd: FILEDESCR)
Pseudocode:

```
Ki(proc, exit()) :
 res = True
side effects:  sid(proc) becomes undef.

Ki(proc, close(fd: FILEDESCR)) :
 res = True
side effects:  sid(fd) becomes undef.
```

The Execve kernel call

Standard signature:
int execve (const char *filename, char *const argv [], char *const envp[]);
Our signature: •execve(p: PATHNAME, f: FILENAME)
Notes:
1. the returned value is ignored
2. char *filename is split into p (the pathname), and f (the actual filename)
3. argv and envp are ignored.
4. In "exists! n in SID" in the pseudocode, n = undef is ok, too.

Pseudocode:

```
Ki(proc, execve(p: PATHNAME, f: FILENAME)) :
 res = (exists! n in SID | n = Sst(sid(proc),sid(f),"process")
                           and (for each directory dir in p
                                Ssav(sid(proc),sid(dir),"dir","search"))
                           and Ssav(sid(proc),sid(f),"file","execute")
                           and Ssav(sid(proc),n,"process","transition")
                           and Ssav(n,sid(f),"process","entrypoint")
                           and Ssav(n,sid(f),"process","execute"))
 act = {close(d) | d in FILEDESCR
                   and d is used by proc
                   and not Ssav(new,sid(d),"fd","inherit") }
side effects :
 if res and not Sst(sid(proc),sid(f),"process") = undef
 then sid(proc) becomes Sst(sid(proc),sid(f),"process").
```

The Kill kernel call

First, a caveat on signals: the set of possible process signals is partitioned into 4 equivalence classes: sigKill, sigstop, sigchld, and signal (the last one represents all the other possible signals). It is still unclear how the different signal permissions are handled. E.g. why do they need *every* signal permission for Killing another process? Is sigKill not enough? And if it's not enough, why did they partition the set of signals?

Standard signature: int kill(pid_t pid, int sig);
Our signature: kill(p: PID)
Notes:
1. int sig and the returned value are ignored.
2. p: PID represents pid_t pid.
3. We present *our* version of kill. The original version = wait (see page 27 of [8]).

Pseudocode:

```
Ki(proc, kill(p: PID)) :
 res = Ssav(sid(proc),sid(p),"process","sigKill").
```

Wait

Standard signature: pid_t wait(int *status)
Our signature: •wait(child: PID)
Notes:
1. int *status is ignored.
2. child: PID represents the returned value.

Pseudocode:

```
Ki(proc, wait(child: PID)) :
 res = Ssav(sid(child),sid(proc),"process","sigKill")
   and Ssav(sid(child),sid(proc),"process","sigstop")
   and Ssav(sid(child),sid(proc),"process","sigchld")
   and Ssav(sid(child),sid(proc),"process","signal").
```

Fork

Standard signature: pid_t fork(void);
Our signature: •fork(p: PID)
Notes:
1. p: PID represents the returned value.

Pseudocode:

```
Ki(proc, fork(child: PID)) :
 res = Ssav(sid(proc),sid(proc),"process","fork")
side effects :  sid(child) becomes sid(proc).
```

Uselib

Standard signature: int uselib(const char *library);
Our signature: uselib(f: FILENAME)
Notes:
1. the returned value is ignored
2. f: FILENAME represents char *library

Pseudocode:

```
Ki(proc, uselib(f: FILENAME)) :
 res = Ssav(sid(proc),sid(f),"process","execute").
```

Ptrace

Standard signature:
int ptrace(int request, pid_t pid, void * addr, void * data)
Our signature: ptrace(p: PID)

Notes:

1. p: PID represents pid_t pid
2. everything else is ignored

Pseudocode

```
Ki(proc, ptrace(p: PID)) :
 res = Ssav(sid(proc),sid(p),"process","ptrace").
```

Getpriority

Standard signature: int getpriority(int which, int who);
Our signature: getpriority(p: PID)
Notes:

1. int which = PRIO_PROCESS, in this case
2. p: PID represents int who
3. the returned value is ignored

Pseudocode:

```
Ki(proc, {getpriority,getscheduler,getparam}(p: PID)) :
 res = Ssav(sid(proc),sid(p),"process","getsched").
```

Setpriority, Setscheduler, Setparam

Standard signature: int setpriority(int which, int who, int prio);
Our signature: •setpriority(p: PID)
Notes:

1. int which = PRIO_PROCESS, in this case
2. p: PID represents int who
3. everything else is ignored

Pseudocode:

```
Ki(proc, {setpriority,setscheduler,setparam}(p: PID)) :
 res = Ssav(sid(proc),sid(p),"process","setsched").
```

Getsid

Standard signature: pid_t getsid(pid_t pid);
Our signature: getsid(p: PID)
Notes:

1. p: PID represents pid_t pid
2. the returned value is ignored
3. getsid gets the session id

Pseudocode:

```
Ki(proc, getsid(p: PID)) :
 res = Ssav(sid(proc),sid(p),"process","getsession").
```

Getpgid, Setpgid

Standard signatures:
pid_t getpgid(pid_t pid); int setpgid(pid_t pid, pid_t pgid);
Our signatures: getpgid(p: PID); •setpgid(p: PID)
Notes:
1. p: PID represents pid_t pid
2. returned values and pid_t pgid are ignored
3. getpgid and setpgid get and set the process group

Pseudocode:

```
Ki(proc, getpgid(p: PID)) :
 res = Ssav(sid(proc),sid(p),"process","getpgid").

Ki(proc, setpgid(p: PID)) :
 res = Ssav(sid(proc),sid(p),"process","setpgid").
```

Process calls without requirements

NOREQ = •get*uid, •get*gid, •getgroups, getitimer, •getpgrp, •getpid, •getppid, getrlimit, getrusage, signal, sigaction, sigalstack, sigprocmask, sigpending, sigsuspend, nanosleep, pause

```
Ki(proc, call : NOREQ) :
 res = True.
```

C.3.2. Plain File Management

CAVEAT: From [8], page 34:

> A file permission check uses the class of the file being accessed, so the *file* class in the tables may be the pipe class, the directory class, or any of the file object classes.

Note: the *file* corresponds to the third argument of Ssav.
Question: pipes are created with the "pipe" call, but this generates two fd (file description classes/SIDs). What are the control requirements?

Open

Standard signature: int open(const char *pathname, int flags);
Our signature:
•open(fld: FILEDESCR, p: PATHNAME, f: FILENAME,
perms: set of {read, write, append})

Notes:

1. p: PATHNAME, f: FILENAME represent char *pathname
2. perms: set of read, write, append represents int flags
3. fld: FILEDESCR represents the returned value
4. The system call described opens an existing file
5. caveat: file creation is in dir management
6. Note: from this definition of sid(fd) in the "side effects" below, it follows that Ssav(x,y,"fd","create") should always be called with x = y. Is this true?

Pseudocode:

```
Ki(proc, open(fld: FILEDESCR, p: PATHNAME, f: FILENAME,
              perms: set of {read, write, append})) :
 res = (for each directory dir in p
                    Ssav(sid(proc),sid(dir),"dir","search"))
               and Ssav(sid(proc),sid(proc),"fd","create")
               and (for each p1 in perms Ssav(sid(proc),sid(f),"file", p1)))
side effects : if res then sid(fld) becomes sid(proc).
```

Read, Readv, Pread, Write, Writev, Pwrite

Standard Signatures:

ssize_t write(int fd, const void *buf, size_t count);
ssize_t read(int fd, void *buf, size_t count);

Our signatures:

{•read, readv, pread,•write, writev, pwrite}(fildes: FILEDESCR,
f: FILENAME)

Notes:

1. f: FILENAME is the open file corresponding to int fd
2. fildes: FILEDESCR represents int fd
3. the other parameters/retured value are ignored

Pseudocode:

```
Ki(proc, {read, readv, pread}(fildes: FILEDESCR, f: FILENAME)) :
 res = Ssav(sid(proc),sid(fildes),"fd","setattr")
       and Ssav(sid(proc),sid(f),"file","read").

Ki(proc, {write, writev, pwrite}(fildes: FILEDESCR, f: FILENAME)) :
 res = Ssav(sid(proc),sid(fildes),"fd","setattr")
   and (Ssav(sid(proc),sid(f),"file","write") or
        Ssav(sid(proc),sid(f),"file","append")).
```

Sendfile

Our signature: sendfile(in_fd, out_fd: FILEDESCR, in_f, out_f: FILENAME)
Pseudocode:

```
Ki(proc, sendfile(in_fd, out_fd: FILEDESCR, in_f, out_f: FILENAME)) :
 res = Ssav(sid(proc),sid(in_fd),"fd","setattr")
   and Ssav(sid(proc),sid(in_f),"file","read")
   and Ssav(sid(proc),sid(out_fd),"fd","setattr")
   and (Ssav(sid(proc),sid(out_f),"file","write") or
        Ssav(sid(proc),sid(out_f),"file","append")).
```

Mmap, Mprotect

Standard signatures:
void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
int mprotect(const void *addr, size_t len, int prot);
Our signatures:
{•mmap, •mprotect}(fildescr: FILEDESCR, f: FILENAME,
perms: set of {read, write, append, execute})
Notes:

1. (mmap) fildescr: FILEDESCR corresponds to int fd
2. (mmap) f: FILENAME is the open file corresponding to int fd
3. perms corresponds to int prot
4. (mprotect) filedescr & f are those of the file "mmapped" at address void *addr
5. side effects for memory? (→ munmap)

Pseudocode:

```
Ki(proc, {mmap, mprotect}(fildescr: FILEDESCR, f: FILENAME,
              perms: set of {read, write, append, execute})) :
 res = Ssav(sid(proc),sid(filedescr),"fd","setattr")
   and (for each p1 in perms-{execute} Ssav(sid(proc),sid(f),"file", p1))
   and (if "execute" in perms
        then Ssav(sid(proc),sid(f),"process", "execute")
        else True).
```

Stat, Fstat, Lstat

Standard Signatures:
int stat(const char *file_name, struct stat *buf);
int fstat(int filedes, struct stat *buf);
Our signatures: {•stat, •fstat, lstat}(p: PATHNAME, f: FILENAME)
Notes:

1. (stat) p: PATHNAME, f: FILENAME represent char *file_name
2. (fstat) p: PATHNAME, f: FILENAME represent the complete pathname of the open file referenced by int filedes

3. remaining items are discarded

Pseudocode:

```
Ki(proc, {stat, fstat, lstat}(p: PATHNAME, f: FILENAME)) :
 res = (for each directory dir in p Ssav(sid(proc),sid(dir),"dir","search"))
       and Ssav(sid(proc),sid(f),"file","getattr").
```

Fchown, Lchown, Chown, Fchmod, Chmod, Ftruncate, Truncate, Utime

Standard signatures:

int chown(const char *path, uid_t owner, gid_t group);
int chmod(const char *path, mode_t mode);
int utime(const char *filename, struct utimbuf *buf);

Our signatures:

{•[f,l]chown, •[f]chmod, [f]truncate, •utime[s]} (p: PATHNAME, f: FILENAME)

Notes:

1. (chown/chmod) p: PATHNAME, f: FILENAME represent char *path
2. (utime) p: PATHNAME, f: FILENAME represent char *filename
3. the other parameters/returned value are ignored

Pseudocode:

```
Ki(proc, {[f,l]chown, [f]chmod, [f]truncate, utime[s]}
  (p: PATHNAME, f: FILENAME)) :
 res = (for each directory dir in p Ssav(sid(proc),sid(dir),"dir","search"))
       and Ssav(sid(proc),sid(f),"file","setattr").
```

Access

Standard signature: int access(const char *pathname, int mode);
Our signature: •access(f: FILENAME)
Notes:

1. f: FILENAME represents char *pathname
2. the other parameters/retured value are ignored

Pseudocode:

```
Ki(proc, access(f: FILENAME)) :
 res = Ssav(sid(proc),sid(f),"file","access").
```

Poll, Select

Our signatures: {poll, select}(f: FILENAME)
Pseudocode:

```
Ki(proc, {poll, select}(f: FILENAME)) :
 res = Ssav(sid(proc),sid(f),"file","poll").
```

Flock, Fcntl

Standard signature: int fcntl(int fd, int cmd);
Our signatures:
flock(f: FILENAME),
•fcntl(f: FILENAME, c: {F_GETLK, F_SETLK, F_SETLKW})

Notes:

1. f: FILENAME corresponds to the open file referenced by int fd
2. c represents int cmd, only the commands F_GETLK, F_SETLK, and F_SETLKW are considered.

Pseudocode:

```
Ki(proc, {flock, fcntl}(f: FILENAME)) :
 res = Ssav(sid(proc),sid(f),"file","lock").
```

Additional fcntl commands

Our signature:
fcntl(f: FILENAME, d: FILEDESCR, cmd: {F_SETFL,F_GETFL,
F_GETOWN,F_GETSIG,F_SETOWN,F_SETSIG})
Pseudocode:

```
Ki(proc, fcntl(f: FILENAME, d: FILEDESCR,
   cmd: {F_SETFL,F_GETFL,F_GETOWN,F_GETSIG,F_SETOWN,F_SETSIG})) :
 res = if cmd = "F_SETFL"
       then Ssav(sid(proc),sid(d),"fd","setattr")
        and Ssav(sid(proc),sid(f),"file","write")
       elsif cmd = "F_SETOWN" or cmd = "F_SETSIG"
       then Ssav(sid(proc),sid(d),"fd","setattr")
       else Ssav(sid(proc),sid(d),"fd","getattr").
```

Lseek

Standard signature: off_t lseek(int fildes, off_t offset, int whence);
Our signature: •lseek(d: FILEDESCR)
Notes:

1. d: FILEDESCR corresponds to int fildes
2. everything else is ignored

Pseudocode:

```
Ki(proc, lseek(d: FILEDESCR)) :
 res = Ssav(sid(proc),sid(d),"fd","setattr").
```

C.3.3. Some ioctl calls... (i/o files access)

Note: From the man page of ioctl:

Arguments, returns, and semantics of ioctl vary according to the device driver in question (the call is used as a catch-all for operations that don't cleanly fit the Unix stream I/O model).

The standard signature of ioctl is:

int ioctl(int d, int request, ...)

where int d is usually an open file descriptor, int request is the name of the ioctl call, and the third argument depends on the name of the call.

In our definitions, we put the name argument of the call (i.e. "request") as sort of tag in the same name of the call, because it ideally is not a parameter. For example the name ioctl_FIBMAP stands for the FIBMAP ioctl call, i.e. with d = FIBMAP.

Ioctl_FIBMAP, Ioctl_FIONREAD

Our signatures:
ioctl_{FIBMAP,FIONREAD,SETFLAGS,SETVERSION}
(f: FILENAME, fd: FILEDESCR)
Pseudocode:

```
Ki(proc, ioctl_FIBMAP(f: FILENAME, fd: FILEDESCR)) :
 res = Ssav(sid(proc),sid(f),"file","ioctl")
   and Ssav(sid(proc),sid(f),"file","getattr").

Ki(proc, ioctl_FIONREAD(f: FILENAME, fd: FILEDESCR)) :
 res = Ssav(sid(proc),sid(f),"file","ioctl")
   and Ssav(sid(proc),sid(fd),"fd","getattr")
   and Ssav(sid(proc),sid(f),"file","getattr").

Ki(proc, ioctl_{SETFLAGS, SETVERSION} (f: FILENAME, fd: FILEDESCR)) :
 res = Ssav(sid(proc),sid(f),"file","ioctl")
   and Ssav(sid(proc),sid(f),"file","setattr").
```

Ioctl calls using a file descriptor

Our signature: {ioctl_FIONBIO, ioctl_FIOASYNC}(d: FILEDESCR)
Pseudocode:

```
Ki(proc, {ioctl_FIONBIO,ioctl_FIOASYNC}(d: FILEDESCR)) :
 res = SAV(sid(proc),sid(d),"fd","setattr").
```

Generic ioctl call

Standard signature: int ioctl(int d, int request, char *argp)
Our signature: •ioctl(f: FILENAME)
Notes:

1. f: FILENAME corresponds to int d, the device descriptor
2. int request is not considered here, but can change the semantics of the ioctl call

3. char *argp depends on int request, therefore is ignored here

Pseudocode:

```
Ki(proc, ioctl(f: FILENAME)) :
 res = Ssav(sid(proc),sid(f),"file","ioctl").
```

C.3.4. Directory Management

Chdir, Fchdir, Chroot

Standard signature: int chdir(const char *path);
Our signatures: {•[f]chdir, chroot}(p: PATHNAME))
Notes:

1. p: PATHNAME corresponds to char *path
2. the return value is ignored

Pseudocode:

```
Ki(proc, {[f]chdir, chroot}(p: PATHNAME)) :
 res = (for each directory dir in p Ssav(sid(proc),sid(dir),"dir","search")).
```

Creat, Open

Standard signature: int open(const char *pathname, int flags);
Our signatures:
{•creat, •open}(fld: FILEDESCR, fls: FILESYSTEM, p: PATHNAME, f: FILENAME, perms: set of {read, write, append})
Notes:

1. creat opens a nonexisting file
2. p: PATHNAME, f: FILENAME represent char *pathname
3. perms: set of read, write, append represents int flags
4. fld: FILEDESCR represents the returned value
5. fls: FILESYSTEM represents the current file system

Pseudocode:

```
Ki(proc, {creat, open}(fld: FILEDESCR, fls: FILESYSTEM,
   p: PATHNAME, f: FILENAME, perms: set of {read, write, append})) :
 res = (exists! n in SID |
        n = Sst(sid(proc),sid(last element of p),"file")
        and Ssav(sid(proc),n,"file","create")
        and Ssav(n,sid(fls),"fs","associate"))
   and (for each directory dir in p Ssav(sid(proc),sid(dir),"dir","search"))
   and Ssav(sid(proc),sid(proc),"fd","create")
   and Ssav(sid(proc),sid(last element of p),"dir","add_name")
   and (for each p1 in perms Ssav(sid(proc),sid(f),"file", p1))
```

```
side effects :
 if res then
  sid(fld) becomes sid(proc)
  and if not Sst(sid(proc),sid(last element of p),"file") = undef
   then sid(f) becomes Sst(sid(proc),sid(last element of p),"file")
   else sid(f) becomes sid(last element of p).
```

Mkdir, Mknod, Symlink

Our signatures:

{mkdir,mknod,symlink}(fls: FILESYSTEM, p: PATHNAME, d: FILENAME)

Notes:

1. mknod - make a character special file
2. symlink - create a symbolic link
3. note: a directory is a file, too

Pseudocode:

```
Ki(proc, mkdir (fls: FILESYSTEM, p: PATHNAME, d: FILENAME)) :
 res =  (exists! n in SID |
          n = Sst(sid(proc),sid(last element of p),"dir")
          and (for each directory dir in p
               Ssav(sid(proc),sid(dir),"dir","search"))
           and Ssav(sid(proc),sid(last element of p),"dir","add_name")
           and Ssav(sid(proc),n,"file","create")
           and Ssav(n,sid(fls),"fs","associate").
side effects: if res then
  if not Sst(sid(proc),sid(last element of p),"dir") = undef
  then sid(d) becomes Sst(sid(proc),sid(last element of p),"dir")
  else sid(d) becomes sid(last element of p).

Ki(proc, mknod (fls: FILESYSTEM, p: PATHNAME, d: FILENAME)) :
 res =  (exists! n in SID |
         n = Sst(sid(proc),sid(last element of p),"chr_file")
         and (for each directory dir in p
              Ssav(sid(proc),sid(dir),"dir","search"))
          and Ssav(sid(proc),sid(last element of p),"dir","add_name")
          and Ssav(sid(proc),n,"file","create")
          and Ssav(n,sid(fls),"fs","associate").
side effects: if res then
  if not Sst(sid(proc),sid(last element of p),"chr_file") = undef
  then sid(d) becomes Sst(sid(proc),sid(last element of p),"chr_file")
  else sid(d) becomes sid(last element of p).
```

```
Ki(proc, symlink (fls: FILESYSTEM, p: PATHNAME, d: FILENAME)) :
 res =  (exists! n in SID |
         n = Sst(sid(proc),sid(last element of p),"lnk_file")
         and (for each directory dir in p
              Ssav(sid(proc),sid(dir),"dir","search"))
          and Ssav(sid(proc),sid(last element of p),"dir","add_name")
          and Ssav(sid(proc),n,"file","create")
          and Ssav(n,sid(fls),"fs","associate").
side effects: if res then
  if not Sst(sid(proc),sid(last element of p),"lnk_file") = undef
  then sid(d) becomes Sst(sid(proc),sid(last element of p),"lnk_file")
  else sid(d) becomes sid(last element of p).
```

Rename

Standard signature: int rename(const char *oldpath, const char *newpath);
Our signature:

- rename(oldp: PATHNAME, oldf: FILENAME,
  newp: PATHNAME, newf: FILENAME)

Notes:

1. oldp: PATHNAME, oldf: FILENAME correspond to char *oldpath
2. newp: PATHNAME, newf: FILENAME correspond to char *newpath
3. return value ignored

Pseudocode:

```
Ki(proc, rename(oldp: PATHNAME, oldf: FILENAME, newp: PATHNAME, newf: FILENAME)):
 res = (for each directory d in oldp Ssav(sid(proc),sid(d),"dir","search"))
   and Ssav(sid(proc),sid(last element of oldp),"dir","remove_name")
   and (for each directory d in newp Ssav(sid(proc),sid(d),"dir","search"))
   and Ssav(sid(proc),sid(oldf),"file","rename")
   and Ssav(sid(proc),sid(last element of newp),"dir","add_name")
   and (if oldf is a directory
        and not (last element of oldp = last element of newp)
        then Ssav(sid(proc),sid(newf),"dir","reparent")
        else True)
   and (if newp/newf exists
        then Ssav(sid(proc),sid(last element of newp),"dir","remove_name")
             and (if newf is a directory
               then Ssav(sid(proc),sid(newf),"dir","rmdir")
               else Ssav(sid(proc),sid(newf),"file","unlink"))
        else True).
```

Link

Standard signature: int link(const char *oldpath, const char *newpath);
Our signature: •link(p: PATHNAME, f: FILENAME)
Notes:
1. p: PATHNAME, f: FILENAME correspond to char *newpath
2. return value ignored

Pseudocode:

```
Ki(proc, link(p: PATHNAME, f: FILENAME)) :
 res = (for each directory d in p Ssav(sid(proc),sid(d),"dir","search"))
   and Ssav(sid(proc),sid(last element of p),"dir","add_name")
   and Ssav(sid(proc),sid(f),"file","link").
```

Unlink

Standard signature: int unlink(const char *pathname);
Our signature: •unlink(p: PATHNAME, f: FILENAME)
Notes:
1. p: PATHNAME, f: FILENAME correspond to char *pathname
2. return value ignored

Pseudocode:

```
Ki(proc, unlink(p: PATHNAME, f: FILENAME)) :
 res = (for each directory d in p Ssav(sid(proc),sid(d),"dir","search"))
   and Ssav(sid(proc),sid(last element of p),"dir","remove_name")
   and Ssav(sid(proc),sid(f),"file","unlink").
```

Rmdir

Our signature: rmdir(p: PATHNAME, f: FILENAME)
Pseudocode:

```
Ki(proc, rmdir(p: PATHNAME, f: FILENAME)) :
 res = (for each directory d in p
        Ssav(sid(proc),sid(d),"dir","search"))
   and Ssav(sid(proc),sid(last element of p),"dir","remove_name")
   and Ssav(sid(proc),sid(f),"dir","rmdir").
```

Getdents, Readdir

Standard signature:
int getdents(unsigned int fd, struct dirent *dirp, unsigned int count);
Our signature: {•getdents, readdir}(fd: FILEDESCR, d: FILENAME)
Notes:
1. fd: FILEDESCR corresponds to int fd
2. d: FILENAME corresponds to dirent *dirp
3. int count & return value are ignored

Pseudocode:

```
Ki(proc, {getdents, readdir} (fd: FILEDESCR, d: FILENAME)) :
 res = Ssav(sid(proc),sid(df),"fd","setattr")
   and Ssav(sid(proc),sid(d),"dir","read").
```

Readlink

Standard signature: int readlink(const char *path, char *buf, size_t bufsiz);
Our signature: •readlink(f: FILENAME)
Notes:

1. f: FILENAME corresponds to char *path
2. everything else is ignored

Pseudocode:

```
Ki(proc, readlink(f: FILENAME)) :
 res = Ssav(sid(proc),sid(f),"file","read").
```

C.3.5. Socket Management

We limit ourselves to Unix datagram sockets. Note: the notation used is analogous to the one for ioctl calls.

Socket_unix_dgram

Standard signature: int socket(int domain, int type, int protocol);
Our signature: •socket_UNIX_DGRAM(so : SOCKET)
Notes:

1. so : SOCKET corresponds to the returned value
2. int domain is PF_UNIX
3. int type is SOCK_DGRAM
4. int protocol is 0

Pseudocode:

```
Ki(proc, socket_UNIX_DGRAM(so : SOCKET)) :
 res = Ssav(sid(proc),sid(so),"socket","create").
```

Connect_unix_dgram

Standard signature:
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
Our signature:
•connect_UNIX_DGRAM(so : SOCKET, p : PATHNAME, f : FILENAME)
Notes:

1. so: SOCKET corresponds to int sockfd

2. p: PATHNAME, f: FILENAME corresponds to the file representing the socket. This information is encoded in the structure sockaddr *serv_addr. addrlen contains the length of the structure in bytes.
3. everything else is ignored

Pseudocode:

```
Ki(proc, connect_UNIX_DGRAM(so : SOCKET,
  p : PATHNAME, f : FILENAME)) :
 res = (for each directory d in p Ssav(sid(proc),sid(d),"dir","search"))
   and Ssav(sid(proc),sid(f),"sock_file","write")
   and Ssav(sid(proc),sid(so),"socket","connect").
```

Send_unix_dgram

Standard signature: int send(int s, const void *msg, int len, unsigned int flags);
Our signature: •send_UNIX_DGRAM (so : SOCKET, msg : STRING)
Notes:

1. so: SOCKET corresponds to int s
2. msg : STRING corresponds to void *msg
3. everything else is ignored

Pseudocode:

```
Ki(proc, send_UNIX_DGRAM(so : SOCKET, msg : STRING)) :
 res = Ssav(sid(proc),sid(so),"socket","write")
   and Ssav(sid(so),sid(any_socket),"socket","sendto")
   and Ssav(sid(so),sid(msg),"socket","send_msg").
```

Recv_unix_dgram

Standard signature: int recv(int s, void *buf, int len, unsigned int flags);
Our signature: •recv_UNIX_DGRAM (so : SOCKET, msg : STRING)
Notes:

1. so: SOCKET corresponds to int s
2. msg : STRING corresponds to the returned value
3. everything else is ignored

Pseudocode:

```
Ki(proc, recv_UNIX_DGRAM(so : SOCKET, msg : STRING)) :
 res = Ssav(sid(proc),sid(so),"socket","read")
   and Ssav(sid(so),sid(any_socket),"socket","recvfrom")
   and Ssav(sid(so),sid(msg),"socket","recv_msg").
```